Chapter 1

Typechecking

So far, our interpretation of the typing judgment $\Gamma \vdash e : A$ has been declarative in the sense that given a triple of $\Gamma$, $e$, and $A$, the judgment answers either “yes” (meaning that $e$ has type $A$ under $\Gamma$) or “no” (meaning that $e$ does not have type $A$ under $\Gamma$). While the declarative interpretation is enough for proving type safety of the simply typed $\lambda$-calculus, it does not lend itself well to an implementation of the type system, which takes a pair of $\Gamma$ and $e$ and decides a type for $e$ under $\Gamma$, if one exists. That is, an implementation of the type system requires not a declarative interpretation but an algorithmic interpretation of the typing judgment $\Gamma \vdash e : A$ such that given $\Gamma$ and $e$ as input, the interpretation produces $A$ as output.

This chapter discusses two implementations of the type system. The first employs an algorithmic interpretation of the typing judgment, and is purely synthetic in that given $\Gamma$ and $e$, it synthesizes a type $A$ such that $\Gamma \vdash e : A$. The second mixes an algorithmic interpretation with a declarative interpretation, and achieves what is called bidirectional typechecking. It is both synthetic and analytic in that depending on the form of a given expression $e$, it requires either only $\Gamma$ to synthesize a type $A$ such that $\Gamma \vdash e : A$, or both $\Gamma$ and $A$ to confirm that $\Gamma \vdash e : A$ holds.

1.1 Purely synthetic typechecking

Let us consider a direct implementation of the type system, or equivalently the judgment $\Gamma \vdash e : A$. We introduce a function typing with the following invariant:

\[
\text{typing}(\Gamma, e, A) = \begin{cases} 
\text{okay} & \text{if } \Gamma \vdash e : A \text{ holds.} \\
\text{fail} & \text{if } \Gamma \vdash e : A \text{ does not hold.}
\end{cases}
\]

Since $\Gamma$, $e$, and $A$ are all given as input, we only have to translate each typing rule in the direction from the conclusion to the premise(s) (i.e., bottom-up), as illustrated in the pseudocode below:

\[
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad \text{Var} \quad \iff \quad \text{typing}(\Gamma, x, A) = \\
\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A. e : A \rightarrow B \rightarrow I} \quad \iff \quad \text{typing}(\Gamma, \lambda x : A. e, A \rightarrow B) = \\
\frac{\text{typing}(\Gamma', e, B) \text{ where } \Gamma' = \Gamma, x : A}{\text{typing}(\Gamma, e, A \rightarrow B) =}
\frac{\text{if } \Gamma \vdash e : A \text{ holds.} \text{ else fail}}{
\text{if } x : A \in \Gamma \text{ then okay else fail}}
\]

It is not obvious, however, how to translate the rule $\rightarrow E$ because both premises require a type $A$ which does not appear in the conclusion:

\[
\frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e \ e' : B} \rightarrow E \iff \\
\text{typing}(\Gamma, e e', B) = \\
\text{if } \text{typing}(\Gamma, e, A \rightarrow B) = \text{okay} \text{ and also typing}(\Gamma, e', A) = \text{okay} \text{ then okay else fail} \text{ where } A = ?
\]
Therefore, in order to return okay, typing \((\Gamma, e, e', B)\) must “guess” a type \(A\) such that both typing \((\Gamma, e, A \to B)\) and typing \((\Gamma, e', A)\) return okay. The problem of guessing such a type \(A\) from \(e\) and \(e'\) involves the problem of deciding the type of a given expression (e.g., deciding the type \(A\) of expression \(e'\)). Thus we need to be able to decide the type of a given expression anyway, and are led to interpret the typing judgment \(\Gamma \vdash e : A\) algorithmically so that given \(\Gamma\) and \(e\) as input, an algorithmic interpretation of the judgment produces \(A\) as output.

We introduce a new judgment \(\Gamma \vdash e \triangleright A\), called an \textit{algorithmic typing judgment}, to express the algorithmic interpretation of the typing judgment \(\Gamma \vdash e : A\):

\[
\Gamma \vdash e \triangleright A \quad \iff \quad \text{under typing context } \Gamma, \text{ the type of expression } e \text{ is inferred as } A
\]

That is, an algorithmic typing judgment \(\Gamma \vdash e \triangleright A\) synthesizes type \(A\) (output) for expression \(e\) (input) under typing context \(\Gamma\) (input). The inference rules for algorithmic typing judgments are as follows:

\[
\begin{align*}
\Gamma \vdash x \triangleright A & \quad \text{Var}_a \\
\Gamma, x : A & \in \Gamma \quad \Gamma \vdash x \triangleright A & \quad \text{Var}_a \\
\Gamma \vdash e \triangleright A \to B & \quad \text{E}_a \\
\Gamma \vdash e \triangleright e' \triangleright B & \quad \text{E}_a \\
\Gamma \vdash \text{true} \triangleright \text{bool} & \quad \text{True}_a \\
\Gamma \vdash \text{false} \triangleright \text{bool} & \quad \text{False}_a \\
\Gamma \vdash e \triangleright \text{bool} & \quad \Gamma \vdash e_1 \triangleright A_1 & \quad \Gamma \vdash e_2 \triangleright A_2 & \quad A_1 = A_2 \quad \text{If}_a \\
\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \triangleright A_1 & \\
\end{align*}
\]

Note that in the rule \(\text{If}_a\), we may not write the second premise as \(\Gamma \vdash e' \triangleright A\) (and remove the third premise) because type \(C\) to be inferred from \(\Gamma\) and \(e'\) is unknown in general and must be explicitly compared with type \(A\) as is done in the third premise. (Similarly for types \(A_1\) and \(A_2\) in the rule \(\text{If}_a\).) A typechecking algorithm based on the algorithmic typing judgment \(\Gamma \vdash e \triangleright A\) is said to be purely synthetic.

The equivalence between two judgments \(\Gamma \vdash e \triangleright A\) and \(\Gamma \vdash e : A\) is stated in Theorem 1.3, whose proof uses Lemmas 1.1 and 1.2. Lemma 1.1 proves soundness of \(\Gamma \vdash e \triangleright A\) in the sense that if an algorithmic typing judgment infers type \(A\) for expression \(e\) under typing context \(\Gamma\), then \(A\) is indeed the type for \(e\) under \(\Gamma\). In other words, if an algorithmic typing judgment gives an answer, it always gives a correct answer and is thus “sound.” Lemma 1.2 proves completeness of \(\Gamma \vdash e \triangleright A\) in the sense that for any well-typed expression \(e\) under typing context \(\Gamma\), there exists an algorithmic typing judgment inferring its type. In other words, an algorithmic typing judgment covers all possible cases of well-typed expressions and is thus “complete.”

**Lemma 1.1 (soundness).** If \(\Gamma \vdash e \triangleright A\), then \(\Gamma \vdash e : A\).

**Lemma 1.2 (completeness).** If \(\Gamma \vdash e : A\), then \(\Gamma \vdash e \triangleright A\).

**Theorem 1.3.** \(\Gamma \vdash e : A\) if and only if \(\Gamma \vdash e \triangleright A\).

\textit{Proof.} Follows from Lemmas 1.1 and 1.2. \qed

### 1.2 Bidirectional typechecking

In the simply typed \(\lambda\)-calculus, every variable in a \(\lambda\)-abstraction is annotated with its type (e.g., \(\lambda x : A. e\)). While it is always good to know the type of a variable for the purpose of typechecking, a typechecking algorithm may not need the type annotation of every variable which sometimes reduces code readability.

As an example, consider the following expression which has type bool:

\[
(\lambda f : \text{bool} \to \text{bool}. f \text{ true}) \lambda x : \text{bool}. x
\]

The type of the first subexpression \(\lambda f : \text{bool} \to \text{bool}. f \text{ true}\) is \((\text{bool} \to \text{bool}) \to \text{bool}\), so the whole expression typechecks only if the second subexpression \(\lambda x : \text{bool}. x\) has type \(\text{bool} \to \text{bool}\) (according to the rule \(\text{E}\)).
Then the type annotation for variable \( x \) becomes redundant because it must have type \( \text{bool} \) anyway if \( \lambda x : \text{bool}. \ x \) is to have type \( \text{bool} \rightarrow \text{bool} \). This example illustrates that not every variable in a well-typed expression needs to be annotated with its type.

A bidirectional typechecking algorithm takes a different approach by allowing \( \lambda \)-abstractions with no type annotation (i.e., \( \lambda x. \ e \) as in the untyped \( \lambda \)-calculus), but also requiring certain expressions to be explicitly annotated with their types. Thus bidirectional typechecking assumes a modified definition of abstract syntax:

\[
\text{expression} \quad e \ ::= \; x \; | \; \lambda x. \ e \; | \; e \; | \; \text{true} \; | \; \text{false} \; | \; \text{if} \; e \; \text{then} \; e \; \text{else} \; e \; | \; (e : A)
\]

A \( \lambda \)-abstraction \( \lambda x. \ e \) does not annotate its formal argument with a type. (It is okay to permit \( \lambda x : A. \ e \) in addition to \( \lambda x. \ e \), but it does not expand the set of well-typed expressions under bidirectional typechecking.) \((e : A)\) explicitly annotates expression \( e \) with type \( A \), and plays the role of variable \( x \) bound in a \( \lambda \)-abstraction \( \lambda x : A. \ e \). Specifically it is \((e : A)\) that feeds type information into a bidirectional typechecking algorithm whereas it is \( \lambda x : A. \ e \) that feeds type information into an ordinary typechecking algorithm.

A bidirectional typechecking algorithm proceeds by alternating between an \textit{analysis} phase, in which it “analyzes” a given expression to verify that it indeed has a given type, and a \textit{synthesis} phase, in which it “synthesizes” the type of a given expression. We use two new judgments for the two phases of bidirectional typechecking:

- \( \Gamma \vdash e \uparrow A \) means that we are checking expression \( e \) against type \( A \) under typing context \( \Gamma \). That is, \( \Gamma \), \( e \), and \( A \) are all given and we are checking if \( \Gamma \vdash e : A \) holds. \( \Gamma \vdash e \uparrow A \) corresponds to a declarative interpretation of the typing judgment \( \Gamma \vdash e : A \).

- \( \Gamma \vdash e \downarrow A \) means that we have synthesized type \( A \) from expression \( e \) under typing context \( \Gamma \). That is, only \( \Gamma \) and \( e \) are given and we have synthesized type \( A \) such that \( \Gamma \vdash e : A \) holds. \( \Gamma \vdash e \downarrow A \) corresponds to an algorithmic interpretation of the typing judgment \( \Gamma \vdash e : A \), and is stronger (i.e., more difficult to prove) than \( \Gamma \vdash e \uparrow A \).

Now we have to decide which of \( \Gamma \vdash e \uparrow A \) and \( \Gamma \vdash e \downarrow A \) is applicable to a given expression \( e \). Let us consider a \( \lambda \)-abstraction \( \lambda x. \ e \) first:

\[
\Gamma \vdash \ldots \vdash \lambda x. \ e \downarrow A \rightarrow B \quad \text{b}\quad \text{or} \quad \Gamma \vdash \ldots \vdash \lambda x. \ e \uparrow A \rightarrow B \quad \text{b}
\]

Intuitively we cannot hope to synthesize type \( A \rightarrow B \) from \( \lambda x. \ e \) because the type of \( x \) is unknown in general. For example, \( e \) may not use \( x \) at all, in which case it is literally impossible to infer the type of \( x! \) Therefore we have to check \( \lambda x. \ e \) against a type \( A \rightarrow B \) to be given in advance:

\[
\Gamma, x : A \vdash \ldots \vdash e \uparrow B \quad \text{b}
\]

Next let us consider an application \( e \ e' \):

\[
\Gamma \vdash \ldots \vdash e' \downarrow B \quad \text{b}\quad \text{or} \quad \Gamma \vdash \ldots \vdash e' \uparrow B \quad \text{b}
\]

Intuitively it is pointless to check \( e' \) against type \( B \), since we have to synthesize type \( A \rightarrow B \) for \( e \) anyway. With type \( A \rightarrow B \) for \( e \), then, we automatically synthesize type \( B \) for \( e' \) as well, and the problem of checking \( e' \) against type \( B \) becomes obsolete because it is easier than the problem of synthesizing type \( B \) for \( e' \). Therefore we synthesize type \( B \) from \( e' \) by first synthesizing type \( A \rightarrow B \) from \( e \) and then verifying that \( e' \) has type \( A \):

\[
\Gamma \vdash \ldots \vdash e \downarrow A \rightarrow B \quad \Gamma \vdash e' \uparrow A \quad \Gamma \vdash e' \downarrow B \quad \text{b}\quad \text{b}
\]

For a variable, we can always synthesize its type by looking up a typing context:

\[
\frac{x : A \in \Gamma \quad \Gamma \vdash x \downarrow A}{\text{Var}_b}
\]
Then how can we relate the two judgments $\Gamma \vdash e \uparrow A$ and $\Gamma \vdash e \downarrow A$? Since $\Gamma \vdash e \downarrow A$ is stronger than $\Gamma \vdash e \uparrow A$, the following rule makes sense regardless of the form of expression $e$:

$$
\frac{
\Gamma \vdash e \downarrow A \\
\Gamma \vdash e \uparrow A
}{
\Gamma \vdash \uparrow_{\downarrow_b} e
}
$$

The opposite direction does not make sense, but by annotating $e$ with its intended type $A$, we can relate the two judgments in the opposite direction:

$$
\frac{
\Gamma \vdash e \uparrow A \\
\Gamma \vdash (e : A) \downarrow A
}{
\Gamma \vdash \downarrow_{\uparrow_b} (e : A)
}
$$

The rule $\uparrow_{\downarrow_b}$ says that if expression $e$ is annotated with type $A$, we may take $A$ as the type of $e$ without having to guess, or “synthesize,” it, but only after verifying that $e$ indeed has type $A$.

Now we can classify expressions into two kinds: introduction expressions $I$ and elimination expressions $E$. We always check an introduction expression $I$ against some type $A$; hence $\Gamma \vdash I \uparrow A$ makes sense, but $\Gamma \vdash I \downarrow A$ is not allowed. For an elimination expression $E$, we can either try to synthesize its type $A$ or check it against some type $A$; hence both $\Gamma \vdash E \downarrow A$ and $\Gamma \vdash E \uparrow A$ make sense. The mutual definition of intro and elim expressions is specified by the rules for bidirectional typechecking:

**intro expression**

\[ I ::= \lambda x. I \mid E \mid (I, I) \mid \text{inl} I \mid \text{inr} I \mid \text{case } E \text{ of } \text{inl} x. I \mid \text{inr} x. I \mid (I : A) \]

**elim expression**

\[ E ::= x \mid E I \mid \text{fst } E \mid \text{snd } E \mid (I : A) \]

As you might have guessed, an expression is an intro expression if its corresponding typing rule is an introduction rule. For example, $\lambda x. e$ is an intro expression because its corresponding typing rule is the → introduction rule $\rightarrow_I$. Likewise an expression is an elim expression if its corresponding typing rule is an elimination rule. For example, $e e'$ is an elim expression because its corresponding typing rule is the → elimination rule $\rightarrow_E$, although it requires further consideration to see why $e$ is an elim expression and $e'$ is an intro expression.

For your reference, we give the complete definition of intro and elim expressions by including remaining constructs of the simply typed $\lambda$-calculus. As in $\lambda$-abstractions, we do not need type annotations in left injections, right injections, abort expression, and the fixed point construct. We use a case expression as an intro expression instead of an elim expression. We use an abort expression as an intro expression because it is a special case of a case expression.

**intro expression**

\[ I ::= \lambda x. I \mid (I, I) \mid \text{inl} I \mid \text{inr} I \mid \text{case } E \text{ of } \text{inl} x. I \mid \text{inr} x. I \mid (I : A) \]

**elim expression**

\[ E ::= x \mid E I \mid \text{fst } E \mid \text{snd } E \mid (I : A) \]

Typing rules for bidirectional typechecking are as follows:

\[
\frac{
x : A \in \Gamma \\
\Gamma \vdash x \downarrow A
}{
\Gamma \vdash \text{Var}_b x
}\]

\[
\frac{
\Gamma, x : A \vdash I \uparrow B \\
\Gamma \vdash \lambda x. I \uparrow A \rightarrow B
}{
\Gamma \vdash \rightarrow_{\downarrow_b} (I : A) \rightarrow B
}\]

\[
\frac{
\Gamma \vdash e \downarrow A \rightarrow B \\
\Gamma \vdash e \uparrow A
}{
\Gamma \vdash \downarrow_{\uparrow_b} (e : A) \rightarrow B
}\]

\[
\frac{
\Gamma \vdash E \downarrow A \rightarrow B \\
\Gamma \vdash E \uparrow A
}{
\Gamma \vdash \rightarrow_{\downarrow_b} (E : A) \rightarrow B
}\]

October 10, 2006
1.3 Exercises

Exercise 1.4. Give typing rules for true, false, and if e then e1 else e2 under bidirectional typechecking.

Exercise 1.5. (λx. x) () has type unit. This expression, however, does not typecheck against unit under bidirectional typechecking. Write as much of a derivation · ⊢ (λx. x) () ⊨ unit as you can, and indicate with an asterisk (*) where the derivation gets stuck.

Exercise 1.6. Annotate some intro expression in (λx. x) () with a type (i.e., convert an intro expression I into an elim expression (I : A)), and typecheck the whole expression using bidirectional typechecking.